An Introduction to Formal Methods for the Development of Safety-critical Applications

Haxthausen, Anne Elisabeth

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

An Introduction to Formal Methods for the Development of Safety-critical Applications

Anne E. Haxthausen
DTU Informatics
Technical University of Denmark
DK-2800 Lyngby
ah@imm.dtu.dk

August 30, 2010

Contents

Preface

This report is a delivery to The Danish Government's railway authority, Trafikstyrelsen, as a part of the Public Sector Consultancy service offered by the Technical University of Denmark.

The purpose of the report is to give the reader an insight into the state-of-the-art of formal methods. The reader is assumed to have some knowledge about software development, but not on formal methods.

The background for the railway authorities' interest in formal methods is the fact that during the next decade a total renewal of the Danish signalling infrastructure is going to take place. Central parts of the new systems will be software components that must fulfill strong safety requirements: in order to get the software certified at the highest Safety Integrity Levels of the European CENELEC standards for railway applications, the software providers are expected to use formal methods.

Acknowledgements I would like to thank Kirsten Mark Hansen, Banedanmark, and Henrik Brogaard for useful comments to a draft of this document.

1 Introduction

Software is being used more and more in almost all aspects of daily life, e.g. in transportation, finance, health care, government, and telecommunications, and the reliability of such software is critical for us, especially when failures may lead to catastrophes where people die or values/money are lost. For instance, when we go by train, it is vital for us that the software controlling the trains is correct such that e.g. train collisions are avoided. As another example, when we use a home banking system to make a bank transaction over the internet, it is vital for us that the software controlling this is correct and secure such that the transaction is executed as we have specified and nobody is able to misuse the data we are sending e.g. to get unintended access to our bank account. Such kind of software is rather complex and it is not an easy task to make it correct. Experience from software development projects also shows that software is often full of bugs leading to delays, cost overrun, usability problems etc. A famous example is the Ariane 5 rocket explosion in 1996 that was due to a software bug (a data conversion of a too large number). To help overcoming such problems, it has been suggested to use *formal methods* in the development of critical systems.

This report provides a general introduction to the state-of-the-art of formal methods for the development of safety-critical systems. It defines what is meant by the term "formal methods" and describes what formal methods can be used for. Examples of industrial applications are also given.

2 Formal methods

What are formal methods: In software engineering, *formal methods* are mathematically based techniques and tools for the synthesis (i.e. development) and analysis of software systems. Formal methods can be applied at various points through the software development cycle. Formal methods can also be used in reverse engineering to model and analyse existing systems. This document will focus on formal methods for the specification of functional requirements and design, and for validation/verification which are the most common forms of use of formal methods.

Why using formal methods: The use of formal methods is motivated by the expectation that, as in other engineering disciplines[1], performing appropriate mathematical modelling and analysis can contribute to the correctness of the resulting product. However, it should be noted that the use of formal methods does not miraculously guarantee correctness, but can be used to increase the level of correctness. Using formal methods in the specification and design phases implies not only that *more flaws are found*, but also that they are found already in these *earlier* phases rather than in the testing or maintenance phases. This is

[1] For instance, before building a bridge, constructional engineers will create a mathematical model in order to analyse whether the bridge will be safe (will not crash) when exposed to weather and traffics.

also an important factor as the cost of repairing flaws is much higher in the later phases than in the earlier phases, cf. e.g. the investigation reported in [LRRA98].

Where are formal methods used: Formal methods are usually only used in the development of safety, business, and mission critical software where the cost of faults is high. In section 5 examples of industrial applications will be given.

Different levels of formal methods: Using formal methods does not necessarily mean that one should make everything formal. The use of formal methods can be more or less elaborate. The use can be classified at three levels according to how formal the specification and verification activities are:

1. formal specification
2. formal specification and semi-formal verification
3. formal specification and formal verification

The choice of which level to use in the development of an application should be decided based on how critical the application is and the available resources (time, money, people having the right skills etc.) One may also choose to use different levels for different components to be developed. For instance one could decide to make a formal specification of the whole system, but only to use formal verification for one critical function. Some standards require the use of formal methods at such specific levels, see section 7.

3 Formal specification

This section first defines what is meant by the term "formal specification" and it describes characteristics and advantages of formal specification. Then examples of important specification styles and specification languages are given, and finally some small formal specification examples are given.

3.1 The notion of formal specification

What is a specification: A *specification* is a description of a product (either to be build or existing). Specifications are used in many different engineering disciplines including software engineering. In software engineering the products that are specified are software. Associated with the notion of a specification, there is the notion of what it means for a product to *satisfy* (fulfill/meet/conform to/be compliant with) its specification.

What is a formal specification: A most common practice in software development is to use informal specifications, i.e. specifications written in natural language or using some diagrams or pseudo code. A complement[2] to informal specification is formal specification. A specification is said to be *formal* if it is expressed in a formal notation or a formal specification language, i.e. a language that has a precise syntax (e.g. given by a BNF grammar) and for which every sentence in the language has a unique mathematical meaning. The underlying mathematical concepts are often simple, e.g. being based on mathematical logic and set theory.

What are formal specifications used for: Formal specifications are, just as informal specifications, used in the analysis and design phases of the software development cycle to record requirements and design decisions, respectively. They can be used as contracts or communication media between customer and developers, and between developers. Besides being used as a base for design and implementation, formal specifications can also be used as a base for generating test cases, for simulation and for formal analysis of the described products in order to predict their behaviour before they are implemented.

Why using formal specifications: The following subsections will further characterize formal specifications as

- being abstract (and subject to refinement),
- being precise, and
- allowing for formal (mathematical) analysis

and explain the major advantages of that.

3.1.1 Abstraction and refinement

Specifications are characterized by being *abstract* in the sense that they omit details that are not relevant for their purpose. For instance, architects use floor plans, as the one shown in Figure 1, as specifications of buildings. The floor plan is an abstraction of a building – for instance it omits details about the building materials. Similarly, in software engineering, specifications omit implementation details about the software they describe. How much is omitted depends on where in the software development cycle the specification is used: generally more is omitted in the earlier phases.

In the requirement analysis phase of the software development cycle a *(requirement) specification* is created. This specification should give a description of the requirements to the software system to be implemented. A good requirements specification describes *what* the software system should fulfil (in the form

[2]It should be stressed that formal specification should not replace informal specification, but complement it. Section 3.1.2 will elaborate on that and also explain how the process of making a formal specification helps improving the informal specification.

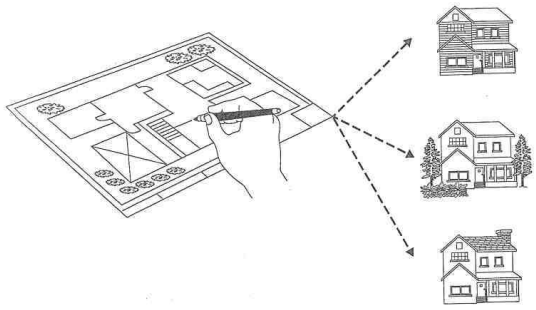

Figure 1: Architects use drawings as specification of buildings.

of some desired properties), and not *how* it should accomplish that. It should concentrate on essential details and defer complicated implementation details to later phases. An example of an (informal) requirement specification of a sorting function *sort* could be:

> The sorting function *sort* should take a sequence of numbers as input and return an ordered sequence of the same numbers.

This specification tells what the result of the sorting should be, but it does not tell how *sort* should sort the numbers. (Later, in example 2, a corresponding formal specification will be shown.)

As a consequence of abstraction, a specification may have *several possible implementations*, i.e. there may be several products that satisfy (i.e. meet or conform to) the specification. For instance, in the architect example, a floor plan may be implemented by several buildings differing with respect to the choice of building materials, as also illustrated in Figure 1. Similarly, for the software example above, the specification of the sorting function has many possible implementations using different sorting algorithms (e.g. quick-sort and insertion sort).

Often, during development of complex products, successively refined specifications are constructed by introducing new aspects and limiting choices. For instance, in the architect example, a floor plan that only specifies the placement of walls, windows and door openings, may be refined into a more detailed floor plan also showing the layers in the walls and the placement of electrical installations, and it may be accompanied by additional drawings showing other construction details and a description of building materials. Similarly, in software engineering, stepwise refinement is used. Starting from the requirement specification (that formalizes the informal requirements), at each step one constructs a more detailed (less abstract/more concrete) description of the system and verifies it against the specification constructed in the previous step. Typically the last specification is a design specification that is so concrete that it is easy to translate it into a programming language. Many formal specification

languages have code generators that can automatically do the translation into selected programming languages. The stepwise refinement process is illustrated in Figure 2. The arrow named `improve` will be explained in section 3.1.2.

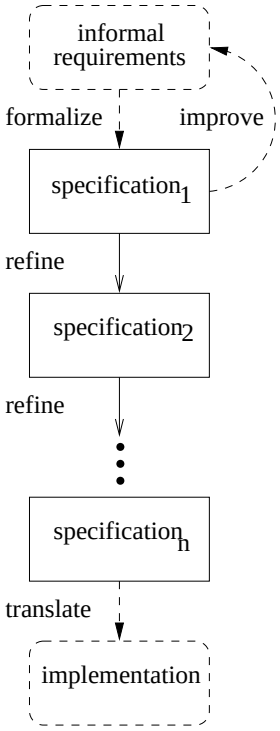

Figure 2: Stepwise refinement of specifications.

3.1.2 Precision

Often specifications are expressed in natural language, as in the example of the sorting function in section 3.1.1, or using some diagrams. However, such informal specifications have the disadvantage that they may be ambiguous, and/or (unintentional) incomplete. For instance, in the sorting example, it is not clear whether the numbers should be ordered with the smallest number first or last. If a formal specification had been used instead, the specifier would have been enforced to make a decision on that and express it in the specification. Generally, the use of mathematics in formal specifications has the advantage that it *enforces the specifier to think deeply about the problem, remembering all possible cases.* Furthermore, it has the advantage that the resulting *formal specifications are unambiguous* as they have a unique mathematical meaning, and hence they are not open to different interpretations. While formal specifications have these advantages of being precise, informal specifications have the advantage of being

more intuitive. Therefore, it is recommended to combine informal specification with formal specification, obtaining the best of each. In the requirement specification phase this is typically done by first capturing and formulating the requirements informally, and then formalizing the informal requirements. Experience also shows that this formalization process often leads to the discovery of ambiguities, incompleteness and inconsistencies in the informal requirements. After the discovery of such problems, they should be resolved so that the formal specification can be made and the informal description of requirements should be changed accordingly as illustrated by the dotted arrow named `improve` in Figure 2. As a consequence also the informal requirements become more precise.

3.1.3 Formal (mathematical) analysis

In contrast to informal specifications, formal specifications can be mathematically analysed as they are mathematically based. Hence, one can use mathematics to prove properties of a formal specification in order to validate the specification and one can verify refinements of specifications.

Verification is the act of investigating whether a product (e.g. a software module) is correct, i.e. satisfies (conforms to) its specification. When stepwise refinement of specifications is used in software development as shown in Figure 2, verification is also done stepwise: in each step one verifies that the new specification satisfies the previous specification. When the specifications are formal, it has the advantage that it is possible to mathematically define what it means for a specification to satisfy another specification[3], and having defined that, it is possible to use mathematics to verify satisfaction in the refinement steps. When mathematics is used in the verification process, it is called *formal verification*. The step from the last specification to the implementation can only be formally verified, if the programming language used for the implementation has been given a formal meaning, but this is only the case for some programming languages such as Ada, and therefore the implementation step (that is often performed by a code generator tool) is usually only informally verified. However, often the last specification is so close to the implementation, that it it is easy to be confident about the informal verification.

Verification is only concerned with the correctness of a product with right to its specification. Another question is whether the specification correctly describes the problem to be solved. *Validation* is the act of investigating the latter. Formal specifications can be validated formally by stating some properties (in the form of some mathematical statements) that are expected to be consequences of the specification, and then using mathematics to prove that. Formal specifications can also be validated by testing when the specifications are executable.

In section 4 *techniques* for formal verification and validation will be explained.

[3]The definition of satisfaction depends on the formal specification language used. Satisfaction basically means that all properties of the previous specification should be consequences of (the properties of) the refined specification.

3.2 Formal specification languages

Many formal specification languages exist. Like for programming languages, there exist different specification languages that are suited for specifying different kinds of systems (e.g. sequential programs or reactive, concurrent systems), and they provide different styles (e.g. model-oriented or property-oriented) for doing this.

3.2.1 Mathematical foundations of formal specification languages

Each formal specification language has its own underlying mathematical logical framework consisting of a notion of (software) *models*, a notion of *sentences* (also called statements or formulas) that can be used to express properties about such models and a notion of what it means for a model to *satisfy* a sentence. The models are mathematical abstractions of programs or systems. Often mathematical structures such as numbers, sets and lists are used to represent data structures, and mathematical functions are used to represent procedures/methods/functions in programs, while so-called transition systems are used to represent reactive systems (for an example of a transition system model, see example 3 in section 4.2.2).

A specification can be a syntactic presentation of a model in which case it is said to be *model-oriented* or it can be a syntactic presentation of collection of sentences (and it then stands for all models that satisfy all the sentences) in which case it is said to be *property-oriented*.

The framework usually also provides *proof rules* for how to construct proofs for a sentence to be a consequence of other sentences or for a model to satisfy a sentence.

3.2.2 Examples of formal specification languages

This section lists examples of notable specification languages that are all characterized by having associated development methods and comprehensive computer-based tool support.

Notable examples of *model-oriented* specification languages include: B [Abr96], Z [WD96], VDM [Jon90, FL09], and VDM++ [FLM+05].

Notable examples of *property-oriented* specification languages include: CASL [MHST08, CoF04, BM04], Maude [CDE+07], CafeOBJ [DF98], and temporal logic languages.

For the specification of concurrent systems there is a class of languages called *process algebras*. The most famous examples of these are CSP [Hoa85] and CCS [Mil80]. Since their invention many specialized process algebras have been invented.

Some languages include several styles. A notable example of this is the RAISE Specification Language, RSL, [GHH+92, GHH+95, GH08] that includes and integrates model-oriented specification, property-oriented specification, and process algebra.

3.3 Small specification examples

This section contains some small examples of formal specifications. It can be skipped by readers not interested in technical details. For the convenience of the reader, appendix A contains an explanation of the mathematical logical operators used in the examples.

Example 1 Consider a railway station having entry signals S1 and S2 for two conflicting routes. It is a requirement that the two signals must never show a go aspect at the same time. A formal specification of this property can be given by the following logical formula:

$$\textbf{always}(\neg(S1 \wedge S2))$$

where $S1$ and $S2$ are Boolean variables that represent the state of the signals such that they are true when the signals show a go aspect and false otherwise. The formula says that it should always (i.e. for all (reachable) states) hold that it is not the case that $S1$ and $S2$ are true (i.e. that the signals are green) at the same time.

Example 2 Below is shown an example of a formal specification of a sorting function `sort` that sorts a list `l` of n integers `l(1)`, ..., `l(n)`. (Here a list is used as a mathematical abstraction that can later be realized by a data structure such as an array in a programming language).

`sort(l :` **Int***`) = l′ :` **Int***
pre `length(l)` $>$ `0`
post `isSorted(l′)` $\wedge$ `isPermutation(l′,l)`

where

- `lenght(l)` is a standard function giving the number of elements in a list `l`
- `isSorted(l′)` $=$
  `(`$\forall$ `i` $\in$ **Int** `:` `1` $\leq$ `i` $\leq$ `length(l′)` $-$ `1` $\Rightarrow$ `l′(i)` $\leq$ `l′(i+1))`
- `isPermutation(l′,l)` $=$ `(`$\forall$ `e` $\in$ **Int** `:` `count(e, l)` $=$ `count(e, l′))`
- `count(e, l)` is a function giving the number of times an integer `e` occurs in a list `l` (can also be formally defined)

In the first line an interface for the function is given. From that one can see that `sort` takes an integer list `l` as argument and gives (returns) a new integer list `l′`. (**Int** is the symbol used for the set of all integers, **Int*** is the set of all lists containing integers, and `l :` **Int*** means `l` belongs to the set **Int***.) No function body is given for the function. Instead a so-called *pre condition* after the keyword **pre** and two so-called *post conditions* after the keyword **post** are given. The pre condition states under which circumstances it is legal to

apply the function, and the post conditions state some properties that should be fulfilled after the function has been applied (legally). The pre condition in this example states that it is only legal to apply `sort` to a list `l` that is not empty (otherwise it is not meaning-full to sort the list). The post condition states that *after* the the `sort` function has been applied to a list `l`, the new list $\texttt{l}'$ will be *sorted* in a numerical ascending order, i.e. it will hold that for any two consecutive indices `i` and `i+1` of the list, $\texttt{l}'\texttt{(i)}$ will be less than or equal to $\texttt{l}'\texttt{(i+1)}$. It also states that the new list $\texttt{l}'$ is a *permutation* of `l`, i.e. for any integer `e` it holds that the number of times `e` occurs in `l` is the same as the number of times `e` occurs in $\texttt{l}'$.

There exist many algorithms satisfying this specification. One of them is the following insertion sort algorithm:

```
sort(l : Int*) =
variable i, j, newValue : Int
begin
for i := 2 to length(l) do
begin
    newValue = l(i);
    j := i;
    while ((j > 1) ∧ (l(j−1) > newValue)) do
    begin
        l(j) := l(j−1);
        j := j−1;
    end;
    l(j) := newValue;
end;
return l;
end
```

It should be possible to prove (or dis-prove) that the list returned by this algorithm actually satisfies the requirements stated by the post condition when applied to a list `l` that satisfies the pre condition. However, it is out of the scope of this document to show such a proof. Instead, section 4.1 discusses general issues concerning possible ways of making proofs.

4 Formal verification

Traditionally, software validation and verification has been done by techniques such as code inspection and testing. Although these are useful techniques to find bugs, they may not find all of them. For testing large systems this is due to the fact that the number of possible system executions is usually so large that it is only feasible to test a small amount of them.

When formal specifications are used in the specification and design phases, it is possible to use *formal verification* techniques.

The advantages of formal verification are two-fold:

- they consider all possible situations, and
- bugs can be found before the system is implemented, i.e. earlier than by implementation testing.

A disadvantage is that the verification is performed on the specifications of the system and not on the system it-self[4]. Therefore, formal verification is usually seen as a *supplement* to testing and not as an alternative.

There are two major state-of-the-art approaches to formal verification: theorem proving and model checking. These will be described and compared in the following sections.

4.1 Theorem proving

Theorem proving has its roots in mathematical logic and means the act of constructing a *mathematical proof* (a convincing mathematical argument) for a mathematical statement to be true. If the act results in a proof, the statement is known to be true and is said to be a *theorem*. If a proof is not found, one can't conclude that the statement is false. It might be the case that the statement is false, but it could also be the case that the statement is actually true, but the person or proof system used to search for a proof was not wise/powerful enough to find a proof.

Mathematical proofs can be classified as follows according to their rigour:

- *Semi-formal proofs*: Semi-formal[5] proofs are written using a mixture of mathematical formulas and natural language, appealing to the intuition of the reader, in the style most often used in math books. You have probably seen such proofs during your school time, e.g. of Pythagoras' theorem relating the lengths of the sides a, b and c of a right-angled triangle. Provided that there are no flaws in a semi-formal proof, it should in principle be possible to convert it into a formal proof.
- *Formal proofs*: Formal proofs are not using natural language, but are expressed in a symbolic language (called a proof language) having a precise syntax. A proof of a mathematical statement in some mathematical logic consists of a sequence of argumentation steps. An argumentation step consists of some premises (i.e. statements that are known to be true) and a conclusion (a new statement) that can be drawn from the premises. (The mathematical logic provides a number of *(proof) rules* for how one can draw conclusions from premises.) The conclusion of one step can be used as a premise in the following step(s). Eventually this leads to a conclusion which is the statement that should be proved.

[4] This is normally the case, but it is also possible in some special cases to perform verification directly on the code.

[5] What in this document is called *semi-formal* proofs, are by some mathematical logicians called *informal* proofs. However, I prefer to call them semi-formal as they are mathematically based.

If the two approaches should be compared, it is much easier and faster to make semi-formal proofs than formal proofs as the semi-formal proofs are essential just sketches of some formal proofs. However, as semi-formal proofs are sketches and natural language is used, there is the risk that they contain flaws. The process of making formal proofs may be computer aided (as explained in section 4.1.1) limiting the risk of flaws even more and making the process faster and easier than if the formal proof should have been constructed by hand, but still the process of making a computer aided formal proof is often time-consuming and requires experience.

4.1.1 Computer based tools

The process of constructing formal proofs can be aided by computer based tools (= computer programs):

- *Proof checkers* are programs that automatically can check whether a postulated proof is actually a correct proof of a given theorem. This is the simplest form of tool and relatively easy to make.
- *Interactive theorem provers* are programs that can be used to interactively construct a correct proof. This is the most common form of tool.
- *Automated theorem provers* are programs that (almost) automatically search for a proof of a given theorem. Such tools are most difficult to create as it is generally computationally hard to find a proof.

Examples of some notable proof checkers are MetaMath [Met] and Mizar [Miz]. Examples of some notable interactive theorem provers are the PVS [PVS], Isabelle/HOL [Isa], ACL2 [ACL] and Coq [Coq]. Examples of some notable automated theorem provers are Prover9 [Pro] and SPASS [SPA].

4.2 Model checking

Model checking [CGP99, BK08] is an automated approach to verify that a model of a (usually concurrent, reactive) finite state system satisfies a formal specification of requirements to the system. In this approach the models describe how the state of the system may evolve over time[6], and the requirements are some constraints on how the state of the system is allowed to evolve over time. Tools that automatically perform model checking are called *model checkers*.

4.2.1 The process of applying model checking

The process of applying model checking is shown in Figure 3. Given a system that should be verified to meet some given informal requirements (some desired properties of the system), the first step is to create a model of the system and to formalize the requirements obtaining a formal specification of these (called

[6] Note, for concurrent, reactive systems, there are usually many different ways in which the state may evolve over time.

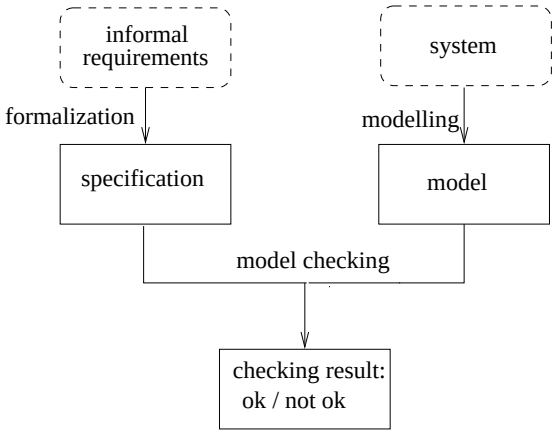

Figure 3: The process of model checking.

a *property specification* in the model checking community). The next step is to use a model checker to check whether the model actually satisfies the property specification. The model checker decides this by exhaustively exploring all system states that can be reached according to the model and check that the property specification holds for these. (The exhaustive exploration is possible as only system models having a finite number of states are considered.) The model checker returns information about whether the model satisfied the property specification or not. In the latter case it will provide a counterexample, i.e. a description of a run of the system (model) that leads to a state for which a the property specification is not meet.

4.2.2 Models

In the model checking approach, a system model describes how the state of the system may change over time. It is typically expressed in terms of so-called finite-state automata (also called finite state machines) that describe the (potential) possible states, the initial state, and the possible state transitions. The models are abstractions that omit details irrelevant for checking the desired properties.

Example 3 The following is an example of a model of a traffic light that turns the red, the yellow, and the green lights on and off in a specific order, as shown in Figure 4. The model is represented in the RSL-SAL specification language [PG07]:

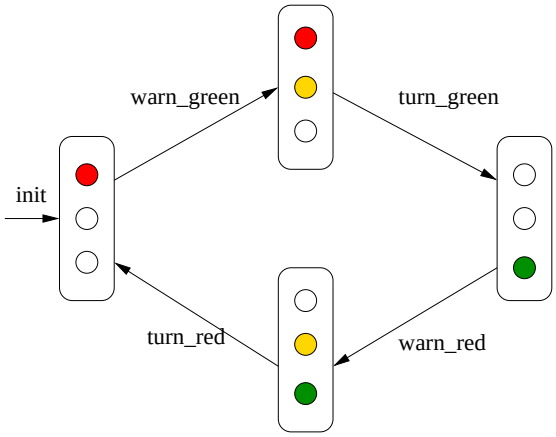

Figure 4: States and state transitions of a traffic light.

```
transition_system [TrafficLightModel]
local
    red :  Bool := true,
    yellow :  Bool := false,
    green :  Bool := false
in
    [warning_green]
      red ∧ ∼yellow ∧ ∼green ⟶ yellow′ = true
    []
    [turn_green]
      red ∧ yellow ∧ ∼green ⟶
        red′ = false ∧ yellow′ = false ∧ green′ = true
    []
    [warning_red]
      ∼red ∧ ∼yellow ∧ green ⟶  yellow′ = true
    []
    [turn_red]
      ∼red ∧ yellow ∧ green ⟶
        red′ = true ∧ yellow′ = false ∧ green′ = false
end
```

In this specification three Boolean variables, `red`, `yellow`, and `green`, are declared. They are, in the following way, used to represent the state of the traffic light: They are **true** when the red, yellow, and green lights are on, respectively, and **false** otherwise. There are potentially 8 states of the system, corresponding to the 8 possible combinations of values of the 3 variables. The variables are initialized to **true**, **false**, and **false**, respectively. This means that the initial state is the one where the red lamp is on and the yellow and the green lamps are

off. There are four transition rules (separated by [] symbols) describing possible changes of the state. The first rule, called `[warning_green]` expresses that when the red light is on and the yellow and green lights are off, it is possible that the yellow light will be turned on. There are three more rules that in a similar way express possible state transitions. In general a transition rule is of the form *precondition* $\longrightarrow$ *postcondition*, where *precondition* and *postcondition* are logical conditions on the states (i.e. on values of the variables). In *postcondition* primed versions of the variables are used. For states where *precondition* is true, a transition into a state where *postcondition* is true may happen. Variables not mentioned in *postcondition* are assumed not to be changed. For instance, in `[warning_green]` the values of the variables `read` and `green` are not changed by the transition.

4.2.3 Property specifications

The property specification is typically expressed in some so-called *temporal logic language* that can be used to express constraints on how the state of a system may evolve over time. Temporal logics are essentially extensions of traditional propositional logic[7] (where the logical formulas contain variables, and logical operators like $\sim$, $\vee$, $\wedge$, and $\Rightarrow$) with operators that refer to the behavior over time. One can for instance use these logics to express

- *safety properties* that express that a system never reaches a bad state (e.g. where two trains collide),
- *lifeness/progress* properties that express that a system will eventually reach a desired state (e.g. one in which a signal shows a go aspect)

The most commonly used temporal logics are Linear Temporal Logic (LTL) and Computational Tree Logic (CTL). For a description of these, see e.g. [CGP99].

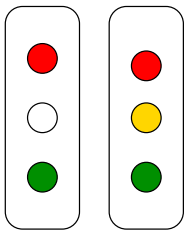

Figure 5: Unsafe states of a traffic light.

Example 4 A desired (safety) property of the traffic light system in example 3 could be that it is never in one of the two bad states shown in Figure 5, where both the red and the green lights are on at the same time. This informally stated property can be formalized by the following assertion:

[7] Propositional logic is called "udsagnslogik" in Danish.

`TrafficLightModel` $\vdash$ **always** `(` $\sim$`(red` $\wedge$ `green)` `)`

This assertion will now be explained. For each possible state the formula $\sim$`(red` $\wedge$ `green)` is either **true** or **false**. For instance, for the initial state the formula is **true**, as in that state `red` = **true** and `green` = **false** such that $\sim$`(red` $\wedge$ `green)` = $\sim$(**true** $\wedge$ **false**) which is equivalent to **true**. The assertion `TrafficLightModel` $\vdash$ **always** `(` $\sim$`(red` $\wedge$ `green))` expresses that $\sim$`(red` $\wedge$ `green)` is **true** in all states that can be reached from the initial state of `TrafficLightModel` using the four transition rules in `TrafficLightModel`. When applying the SAL model checker [SAL01] to check this assertion, it will return with the information that the assertion is true.

4.2.4 Computer based tools

A *model checker* is a computer based tool that automatically performs model checking. Many model checkers exist. Each of these uses a specific language for expressing models and a specific language for expressing properties, and they are implemented using specific model checking techniques/algorithms. Some tools offer the user the choice between different kinds of model checking techniques. Some notable examples of model checkers are SPIN [SPI], NuSMV [NuS], and SAL [SAL01].

4.3 Model checking versus theorem proving

In this section the usability of model checking and theorem proving will be compared.

Model checking has the advantage over theorem proving, that it is fully automated and thereby much easier and faster to use. However, model checking has the disadvantage that it may not be feasible to use for checking large systems due to the so-called *state space explosion problem*, where the number of states needed to model the system exceeds the amount of available computer memory. Recent years many techniques have been invented in order to increase the size of systems that can be model checked without problems, and thus state-of-the-art model checkers can now handle state spaces of about 10^8 to 10^9 states, cf. [BK08]. In contrast to theorem proving, model checking can't be used for checking generalizations such as generic/parametrized systems. As an example, consider a generic railway control system that can be instantiated with application data. Theorem proving can be used to prove correctness of the generic software, i.e. it can be proved *once-and-for-all* that *any* instance of the system, obtained by configuring the generic system with (correct) concrete application data, is correct. This has for instance been done in [HP00, LVH00, GH03]. When using model checking it is not possible to do this, but one must check correctness of each concrete system. This has for instance been done in [HPK09] and [HBK10].

In summary, on one hand model checking is faster and easier to use than theorem proving, and on the other hand theorem proving is applicable in some

cases where model checking isn't due to state space explosion or because the system under consideration can't be modelled as a finite state system model used by model checkers. As a conclusion, a natural choice would be to use model checking whenever this is possible, and theorem proving otherwise.

A current trend in research is to investigate how one can combine the best of model checking with the best of theorem proving, see for instance [RSS95].

5 Examples of industrial usage

In [WLBF09] a survey of industrial use of formal methods in 62 projects showed that the largest single application domain was transport (16%), followed by the financial sector (12%). Below examples of industrial use of formal methods for railways, avionics, and finance, will be given.

5.1 Railways

Formal methods have been applied to railway systems in many countries. Below, first some famous examples of use in the French railway industry will be given, and then examples of case studies made for Danish systems will be given.

5.1.1 RER line A in Paris

Application: In France the first industrial use of formal methods for railways was for the SACEM system of RER Line A in Paris which has been in full operation since 1989. The SACEM system is an automatic train protection system that continuously controls the speed of all trains on the line. The system permanently ensures the safety of 0.8 million passengers per day!

Development: SACEM was developed by GEC Alsthom Transport, MATRA Transport (now Siemens Transportation Systems) and CSEE Transport (now part of Ansaldo) for RAPT in cooperation with SNCF. The software was implemented in the Modula 2 programming language. A combination of many techniques, including formal methods, were used to validate the safety-related software. Formal methods were used in the following way (cf. [GH90, HG93, BDM98]):

1. A formal specification of the functional requirements were made in the B language [Abr96].
2. Pre and post conditions and loop invariants were formulated for the procedures of the source code, and they were proved to hold.
3. It was manually verified that the pre and post conditions satisfied the functional requirements.

All the safety principles were approved by a French national committee for safety composed of members from different ministry and specialists in railways, signalling, safety, and computer science.

Achievements: The formal methods work helped to make the informal requirement specifications more precise (12 differences between the informal specification and the implementation were found by the use of formal methods), cf. [GH90].

The confidence achieved by the use of formal methods for the SACEM system convinced RATP to demand the use of formal methods for the next tender (metro line 14 described below).

Further reading: For more information, see [GH90, HG93].

5.1.2 Metro line 14 in Paris

Application: Another industrial use of formal methods in Paris was for the automatic train operation system for metro line 14 (the first driverless metro line in Paris) that has been in full operation since 1998.

Development: Matra Transport International developed the system for RATP. The B formal method was used to develop and validate safety-critical parts of the automatic train protection system. The safety-critical parts concerned the running and stopping of trains, and the opening and closing of the train doors and platform doors.

The B method and its associated tool kit were industrialized by Matra Transport, RAPT and Stéria Méditérrannée for the purpose of this development. As part of this work a very careful preparation was done to define how the formal method could be integrated into an existing organisation.

The formal development started by making an abstract model/specification that formalized the informal requirements. This model was then refined into a concrete design model that later was translated into the programming language ADA. During the development safety properties and other proof obligations were verified by automatic and interactive proof tools.

Achievements: Many errors were found during the proof activities. As a result of the careful use of formal methods no bugs were found during the testing of the system (neither during the functional validation on the host computer, the integration validation on the target computer nor during the on-site testing) and no bugs have been found since the line has been in operation[8]. The testing costs were also reduced because no unit tests were needed. The formal development was cost effective: in particular, the initial budgets were kept.

Further reading: For more information, see [BDM98, BBFM99].

[8] This was stated in [LSP07] which was published in 2007.

5.1.3 Roissy Charles de Gaulle airport shuttle in Paris

Application: The B formal method was used to develop and validate safety-critical parts of the Roissy Airport shuttle (a driverless light train) that has been in full operation since 2007.

Development: Siemens Transportation Systems (formerly Matra Transport) had the responsibility of the development of the system and subcontracted the development of the safety-critical parts to ClearSy who used the B formal method.

Achievements: Similar to those for metro line 14.

Further reading: For more information, see [BA05].

5.1.4 Railway systems in Denmark

The Technical University of Denmark has in collaboration with Kirsten Mark Hansen from Banedanmark, made formal models and verification for a number of case studies of existing interlocking systems in Denmark.

Applications: Three kinds of interlocking systems have been explored:

- Computer based interlocking systems for stations like Snoghøj and Taulov.
- Computer based interlocking systems for lines like Langå-Stevnstrup.
- Relay interlocking systems for stations like Stenstrup.

Development: In all three cases the RAISE formal method [GHH+92, GHH+95, GH08] was used to model the systems and verify safety properties like no derailings or collisions of trains can happen.

In the first two cases the models were parametrized wrt. the network topology of stations and lines, respectively, and the safety properties were verified *once-and-for-all* to hold for *any* instance of these generic models, i.e. for any models obtained by instantiating the generic models with a concrete network topology of a station or line, respectively. The verification was done partly by making semi-formal proofs by hand and partly by making formal proofs using the RAISE interactive theorem prover.

For the relay interlocking systems, model checking was used for verifying Stenstrup. In addition to the safety properties mentioned above, safety properties that could be derived from the train route tables and circuit diagrams were verified too. All together 142 desired properties were verified automatically.

Achievements: In each of the three cases the verification showed that the desired properties hold.

Further reading: For more information on these case studies, see [LVH00], [GH03] and [HBK10], respectively.

5.2 Avionics

Avionics is a major application area of formal methods. This is exemplified by NASA[9] which has about 40 researchers employed in four formal methods groups. These researches evolve and use formal methods dedicated to the development and validation of avionics software at NASA. Below an example of how NASA is using formal methods will be given.

5.2.1 Remote Agent on NASA's Deep Space 1 mission

Application: NASA used formal methods to verify a component of the Remote Agent software. Remote Agent is the first artificial intelligence control system to control a spacecraft without human supervision. It was one out of twelve technologies that were tested by Deep Space 1, a spacecraft dedicated to testing high risk technologies in deep space to lower the cost and risk to future science-driven missions that use them for the first time. DeepSpace 1 was launched in 1998.

Development and achievements: In 1997, before the launching of Deep Space 1, a component of the Remote Agent software was verified by the model checking approach using the SPIN[SPI] model checker. The verification work detected five concurrency errors in the LISP code that the developers acknowledged would not have been found during testing.

In 1999, after the launching of Deep Space 1, a deadlock occurred in another component of the Remote Agent that had *not* been subject to the verification in 1997. This error happened in space within 24 hours of operation, but had not been discovered in over 300 hours of system level testing at NASA's flight system testbed. Formal methods researchers were then asked to model check the problematic component to see whether they could find the reason for the deadlock. In a short amount of time they found out that the error was the same as one of the five errors that were found in 1997 in the first component.

Further reading: For more information, see [HLP+00].

5.3 Finance

Formal methods are relevant for financial applications that are business critical. Below an example of how formal methods have been applied to software for Mondex smart cards will be given.

[9] National Aeronautics and Space Administration is an Executive Branch agency of the United States government, responsible for the nation's civilian space program and aeronautics and aerospace research.

5.3.1 Mondex smart cards

Application: Formal methods have been applied in the development of Mondex, an electronic purse hosted on a smart card. Each card stores financial value (equivalent to cash) as electronic information on a micro chip and provides operations for making financial transactions with other cards via a communication device.

Development: Mondex was developed in 1996 by a consortium led by NatWest, a UK high-street bank. As it was crucial that the cards would be secure, it was decided that Mondex should be certified to the UK standard for high-assurance systems ITSEC, at its highest level, E6 (which is equivalent to Common Criteria Level EAL7). This level requires formal methods to be applied. To satisfy this requirement the software house Logica, supported by the University of Oxford, constructed formal models of the system and its abstract security policy in the Z notation [WD96], accompanied by hand-written proofs that the system design possessed the required security properties.

Achievements: The use of formal methods revealed a bug in the implementation of a secondary protocol, which was then fixed. In the system testing, also required to achieve ITSEC level E6, no bugs were found in those parts that had been subject to formal methods. As a result of the use of formal methods, testing etc., in 1999 Mondex achieved the required certification at ITSEC level E6 (and was actually the first product to achieve this.)

Further reading: For more information on this original work, see [WSC+08]. In 2006 eight international research groups used different languages and tools to investigate the degree of automation that can now be achieved in the correctness proofs. The results of six of these investigations can be found in [FAC08].

6 Cost-effectiveness

Several investigations on cost-effectiveness have been made during the years. A general view is that using formal methods makes the resulting software more correct, and that the costs tend to be increased early in the development life cycle, but reduced later. Most recently, a survey [WLBF09] of the use of formal methods in 62 industrial projects that had employed formal methods was made based on questionnaires. This survey shows that the effect on development time, cost, and quality of the resulting product was generally positive: Three times as many reported a reduction in time, rather than an increase. Five times as many reported a reduction in cost, rather than an increase. In 92% of the cases the quality was improved.

7 Formal methods in certification standards

Several standards for the industrial development of safety-critical software recommend or even require the use of formal methods for the highest software safety integrity levels. Examples of such standards are: CENELEC EN50128 for railways, the Common Criteria and UK Level E6 for Information Technology Security Evaluation, DO-178B Level A for avionics, and UK MoD software Defence Standards 00-55 and 00-56 . Below, CENELEC EN 50128 for railways will shortly be explored. For further reading on formal methods in certification standards, see [Bow93].

7.1 CENELEC EN 50128

CENELEC EN 50128 is a European standard specifying procedures and technical requirements for the development of software for railway control and protection systems. The standard states which techniques are required, highly recommended, recommended, not recommended, or even forbidden in the development process in order to provide software which meets the demands for safety integrity at five different levels.

For the two highest safety integrity levels, EN 50128 highly recommends to use formal methods for:

- software requirements specification,
- software design,
- verification, and
- software validation.

8 Sources of information

8.1 Conferences

There are many conferences concerning formal methods and development of safety critical systems. For the railway industry the following are especially interesting:

- **FORMS/FORMAT – The Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems:** FORMS/FORMAT is a series of symposia that offer scientists facing formal techniques, practitioners and managers, developers and consultants of automotive and railway industries as well as traffic system operators with interest in formal methods a platform for the exchange of scientific experience and the transfer of practical description means, methods and tools for complex automation systems. The next symposium will take place 2nd-3rd of December 2010 in Braunschweig, for further information see `http://www.forms-2010.de/home.html`.

- **SafeComp – The International Conference on Computer Safety, Reliability and Security:** SafeComp is an annual event covering the state-of-the-art, experience and new trends in the areas of computer safety, reliability and security regarding dependable application of computer systems. The conferences provide a platform for knowledge and technology transfer between academia, industry and research institutions. The next event will take place 14 - 17 September 2010 in Vienna, for further information see `http://www.ocg.at/safecomp2010/`.

- **Industry days of FM International Symposium on Formal Methods:** The FM symposia on Formal Methods are held approximately every 18 months and include each an industry day dedicated to industrial experience with the application of Formal Methods. The next symposium will take place 20-24 June 2011, for further information see `http://www.fmeurope.org/?p=340`.

8.2 Further reading

- Proceedings of the above mentioned conferences.

- Proceedings of the FMERail[10] workshops 1998-1999.

- A survey of results and trends in using formal techniques for the development of software for transportation systems, see [Bjø03].

- Survey and experience papers concerning formal methods in general: see for instance [GCR93, CW96, WLBF09].

- Other references in the reference list of this document.

[10] FMERail was an ESPRIT project promoting the adoption of formal methods in the railway domain. The approach of the project was to arrange a series of workshops to show how different formal method technologies can be applied to railway problems.

A Some operators of mathematical logic

This appendix gives a short reminder about logical operators for readers that have once learnt about these.

The following table show symbols used for operators of classical mathematical (propositional) logic.

symbol	math operator
$\wedge$	and
$\vee$	or
$\Rightarrow$	implies
$\sim$	not
$\forall$	for all

The meaning of the four first operators are given by the truth tables below.

$\wedge$	**true**	**false**
true	**true**	**false**
false	**false**	**false**

$\vee$	**true**	**false**
true	**true**	**true**
false	**true**	**false**

$\Rightarrow$	**true**	**false**
true	**true**	**false**
false	**true**	**true**

b	$\sim$ b
true	**false**
false	**true**

References

[Abr96] J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ACL] ACL2, home page: `http://userweb.cs.utexas.edu/users/moore/acl2/`.

[BA05] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In *ZB 2005: Formal Specification and Development in Z and B*, number 3455 in Lecture Notes in Computer Science. Springer, 2005.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A Successful Application of B in a Large Project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science, pages 369–387. Springer-Verlag, 1999.

[BDM98] Patrick Behm, Pierre Desforges, and Jean-Marc Meynadier. Météor: An Industrial Success in Formal Development. In Didier Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, number 1393 in Lecture Notes in Computer Science. Springer, 1998.

[Bjø03] Dines Bjørner. New Results and Current Trends in Formal Techniques for the Development of Software for Transportation Systems. In *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003), Budapest/Hungary*. L'Harmattan Hongrie, May 15-16 2003.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS Vol. 2900 (IFIP Series). Springer, 2004. With chapters by Till Mossakowski, Donald Sannella, and Andrzej Tarlecki.

[Bow93] Jonathan Bowen. Formal methods in safety-critical standards. In *Software Engineering Standards Symposium (SESS'93)*, pages 168–177. Brighton, UK, IEEE Computer Society Press, Aug–Sep 1993.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyne Talcott. *All About Maude - A High-Performance Logical Framework*. Lecture Notes in Computer Science. Springer, 2007.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS Vol. 2960 (IFIP Series). Springer, 2004. With chapters by Hubert Baumeister and Maura Cerioli and Anne Haxthausen and Till Mossakowski and Peter D. Mosses and Donald Sannella and Andrzej Tarlecki and ...

[Coq] Coq, home page: `http://coq.inria.fr/`.

[CW96] Edmund M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, 1996.

[DF98] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report : The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. Amast Series in Computing. World Scientific Publishing, 1998.

[FAC08] Special issue on the Mondex challenge. *Formal Aspects of Computing*, 20(1), 2008.

[FL09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009.

[FLM+05] J. S. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-Oriented Systems*. Springer-Verlag, London, 2005.

[GCR93] S. Gerhart, D. Craigen, and T. Ralston. Observations on industrial practice using formal methods. In *Proceedings of the 15th International Conference on Software Engineering*, pages 24–34. IEEE Computer Society Press, April 1993.

[GH90] G. Guiho and Claude Hennebert. SACEM Software Validation. In *ICSE '90: Proceedings of the 12th international conference on Software engineering*, pages 186–191, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[GH03] Torben Gjaldbæk and Anne E. Haxthausen. Modelling and Verification of Interlocking Systems for Railway Lines. In *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems*. Elsevier Science Ltd, Oxford, 2003. ISBN 0-08-044059-2.

[GH08] Chris George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. In *Logics of Specification Languages*, EATCS. Springer, 2008.

[GHH+92] Chris George, Peter Haff, Klaus Havelund, Anne E. Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall, 1992.

[GHH+95] Chris George, Anne E. Haxthausen, Stephen Hughes, Robert Milne, Sø Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioners Series. Prentice-Hall, 1995.

[HBK10] Anne E. Haxthausen, Marie Le Bliguet, and Andreas A. Kjær. Modelling and Verification of Relay Interlocking Systems. In Christine Choppy and Oleg Sokolsky, editors, *15th Monterey Workshop: Foundations of Computer Software, Future Trends and Techniques for Development*, number 6028 in Lecture Notes in Computer Science. Springer, 2010. Invited paper.

[HG93] Claude Hennebert and G. Guiho. SACEM: A Fault-Tolerant System for Train Speed Control. In *Proc. 23rd Int. Conf. on Fault-Tolerant Computing (FTCS-23)*, pages 624–628, Toulouse, France, 1993. IEEE Computer Society Press.

[HLP+00] Klaus Havelund, Mike Lowry, Seungjoon Park, Charles Pecheur, John Penix, Willem Visser, and Jon L. White. Formal Analysis of the Remote Agent Before and After Flight. In *The Fifth NASA Langley Formal Methods Workshop*, Virginia, June 2000.

[Hoa85] C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.

[HP00] Anne E. Haxthausen and Jan Peleska. Formal Development and Verification of a Distributed Railway Control System. *IEEE Transaction on Software Engineering*, 26(8):687–701, 2000.

[HPK09] Anne E. Haxthausen, Jan Peleska, and Sebastian Kinder. A Formal Approach for the Construction and Verification of Railway Control Systems. *Formal Aspects of Computing*, online first 2009. Special issue in Honour of Dines Bjørner and Zhou Chaochen on Occasion of their 70th Birthdays.

[Isa] Isabelle, home page: `http://isabelle.in.tum.de/`.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[LRRA98] Peter Liggesmeyer, Martin Rothfelder, Michael Rettelbach, and Thomas Ackermann. Qualitätssicherung software-basierter technischer systeme - problembereiche und Lösungsansätze. *Informatik Spektrum*, 21(5):249–258, 1998.

[LSP07] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal Methods in Safety-Critical Railway Systems. In *10th Brasilian Symposium on Formal Methods*, 2007.

[LVH00] Morten P. Lindegaard, Peter Viuf, and Anne E. Haxthausen. Modelling Railway Interlocking Systems. In *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany*, pages 211–217, 2000.

[Met] MetaMath, home page: `http://us.metamath.org/mpegif/mmset.html`.

[MHST08] Till Mossakowski, Anne Haxthausen, Donald Sannella, and Andrzej Tarlecki. CASL, the Common Algebraic Specification Language. In *Logics of Specification Languages*, EATCS. Springer, 2008.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, Berlin, Germany, 1980.

[Miz] Mizar, home page: `http://www.mizar.org/`.

[NuS] NuSMV: a new symbolic model checker, home page: `http://nusmv.fbk.eu/`.

[PG07] Juan Ignacio Perna and Chris George. Model Checking RAISE Applicative Specifications. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, 2007*, pages 257–268. IEEE Computer Society Press, 2007.

[Pro] Prover9, home page: `http://www.cs.unm.edu/~mccune/prover9/`.

[PVS] PVS Specification and Verification System, home page: `http://pvs.csl.sri.com/`.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. pages 84–97. Springer-Verlag, 1995.

[SAL01] Symbolic Analysis Laboratory, SAL, home page: `http://sal.csl.sri.com`, 2001.

[SPA] SPASS, home page: `http://www.spass-prover.org/`.

[SPI] ON-THE-FLY, LTL MODEL CHECKING with SPIN, home page: `http://spinroot.com/spin/whatispin.html`.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

[WSC+08] Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The Certification of the Mondex Electronic Purse to ITSEC Level E6. *Formal Asp. Comput*, 20(1):5–19, 2008.